View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Software institutes' Online Digital Archive

A Self-stabilizing Network Size Estimation Gossip Algorithm for Peer-to-Peer Systems

Ali Ghodsi[1], Sameh El-Ansary[2], Supriya Krishnamurthy[2], and Seif Haridi[1]

SICS Technical Report T2005:16
ISSN 1100-3154
ISRN:SICS-T–2005/16-SE

Keywords: Network Size Estimation, Gossip Algorithms, Peer-to-Peer, DHTs

[1] Royal Institute of Technology (KTH)
{aligh,seif}@kth.se
[2] Swedish Institute of Computer Science (SICS)
{sameh,supriya}@sics.se

Abstract. We present a self-stabilizing network size estimation gossip algorithm which determines the number of nodes in a structured peer-to-peer system. The algorithm can handle joins, leaves, and failures and is applicable to most structured peer-to-peer systems providing a distributed hash table abstraction. Furthermore, the algorithm is self-stabilizing with respect to the local estimates of any node, which might be arbitrary at any given time. Once state corruption ceases, the algorithm eventually adjusts all estimates to the correct value even in presence of joins and leaves. The algorithm only assumes that the system is weakly fair, and does hence not require the nodes to make the same number of exchanges, to be correct.

1 Introduction

Structured peer-to-peer systems such as [**?**,**?**,**?**] have received much attention by the research community recently. These systems are typically very scalable and the number of nodes in the system immensely varies. The network size is, however, a global variable which is not accessible to individual nodes in the system as they only know a subset of the other nodes in the system. This information is, nevertheless, of great importance to these systems, as it can be used to tune the rates which the topology is maintained (by so called periodic stabilization). Moreover, it can be used for load-balancing purposes, and for designing algorithms that adapt their actions depending on the system size[**?**].

In this paper, we suggest a gossip algorithm which every node in the system continuously runs to estimate the total number of nodes in the system. The algorithm always determines the exact number of nodes in the

system as soon as the perturbations (joins, leaves, and failures) stop. As peer-to-peer systems are intended to run continuously, and an incorrect estimate might diverge all future estimates. Therefore, we provide a means to self-stabilize our estimates such that the local estimate of nodes may be corrupt, but the system still converges to a closure which it stays.

1.1 Outline

In Section 2 we give definitions, assumptions and the notation to be used. Thereafter, Section 3 shows the simple gossip algorithm used to estimate the network size. It is also shown how the algorithm can be extended to handle crash failures, and how it can be implemented on-top of existing structured peer-to-peer systems. In Section 4 we extend the algorithm to use self-stabilizing waves to make it self-stabilizing. Finally, we conclude.

2 Definitions and Assumptions

In this section we outline the basic assumptions underlying our algorithm, and also give the definitions to be used.

We assume our structured peer-to-peer system consists of nodes which have unique identifiers belonging to a ring of identifiers $\mathcal{I} = \{0, 1, \cdots, N-1\}$ for some large constant N (not to be confused with the number of nodes). This is general enough to encompass many existing structured peer-to-peer systems such as Chord[**?**], Pastry[**?**], DKS[**?**], Tapstry[**?**] and many others.

For two arbitrary identifiers $x, y \in \mathcal{I}$, we use $x \oplus y$ (resp. $x \ominus y$) for the addition (resp. subtraction) modulo N.

Every node has a pointer to its successor and predecessor on the ring. Formally, we denote the identifiers of the nodes in the system by $\mathcal{P} \subseteq \mathcal{I}$. The function $succ_{\mathcal{P}} : \mathcal{I} \to \mathcal{P}$ is defined as:

$$succ_{\mathcal{P}}(n) = n \oplus min\{i \ominus n | i \in \mathcal{P} \wedge i \neq n\}$$

The predecessor function is similarly defined as:

$$pred_{\mathcal{P}}(n) = n \oplus max\{i \ominus n | i \in \mathcal{P}\}$$

As an example, a ring of size 1024 containing the nodes $\mathcal{P} = \{10, 235, 903\}$, we have that $succ_{\mathcal{P}}(10) = 235$, $succ_{\mathcal{P}}(903) = 10$, $pred_{\mathcal{P}}(235) = 10$, and $pred_{\mathcal{P}}(10) = 903$.

We will assume that there exists an out-of-bound mechanism to make all of the predecessor and successor pointers correct. This can, for example, be achieved by using periodic stabilization[**?**].

Most structured peer-to-peer systems provide a so called distributed lookup service which, given an identifier, finds a node responsible for that identifier. We will assume, without loss of generality, that the lookup algorithm returns the successor of the identifier, i.e. $lookup(i) = succ_{\mathcal{P}}(i)$.

In the algorithms we assume a distributed system modeled by a set of nodes communicating by message passing through a communication network that: (i) is connected, (ii) is asynchronous, (iii) is reliable, and (iv) provides FIFO communication. We shall use the same formal notation as in [**?**] to describe algorithms.

3 The Network Size Estimation Algorithm

We now outline the requirements posed on the network size estimation algorithm, and thereafter present the algorithm.

3.1 Requirements

We pose the following requirements on a solution to the problem: i) It should be able to handle crash failures. ii) It should be *accurate*, i.e. whenever node joins, leaves, and failures stop occurring, every node's estimation of the network size should converge to the exact number of nodes in the system. iii) Nodes might interact at different rates, i.e. some nodes might, because of the asynchrony, interact with other nodes at a much higher rate than others. We only require the network to be weakly fair[**?**].

Furthermore, in Section 4, we will pose the additional requirement that the algorithm should be self-stabilizing with respect to the estimates of the nodes. I.e., the estimate of any node might be arbitrary, but when perturbations stop, the estimates should converge to the optimum and stay there.

3.2 The Algorithm

Our goal is to make an algorithm where each node tries to estimate the average inter-node distance, δ, on the identifier space, i.e. the average distance between two consecutive nodes on the ring. Given a correct value of δ, the number of nodes in the system can be estimated from the size of the identifier space N by $\frac{N}{\delta}$.

Every node $n \in \mathcal{P}$ in the system will keep a local estimate of the average inter-node distance in a local variable d_n. Hence, our goal is to compute $\sum_{i \in \mathcal{P}} \frac{d_i}{|\mathcal{P}|}$.

The philosophy underlying our algorithm is the observation that at any time the following invariant should always be satisfied:

$$N = \sum_{i \in \mathcal{P}} d_i$$

We achieve this by letting each node $i \in \mathcal{P}$ initialize its estimate d_i to its distance to its successor in the identifier space. In other words, $d_i = succ_{\mathcal{P}}(i) \ominus i$. Note that if the system only contains one node, $d_i = N$. Clearly, an correctly initialized network satisfies the mentioned invariant as the sum of the estimates is equal to the size of the identifier space, N.

As seen by Algorithms 1 and 3, a joining node n initializes its estimate to the distance to its successor, and in addition sends its estimate, d_n, to its successor, s, which subtracts d_n from its local estimate d_s. Therefore, the arrival of a node makes sure that the invariant of the sum of the inter-node distances is preserved.

Similarly, a node l leaving the network sends a message with its current estimate d_l to its successor s, which then adds d_l to its local estimate d_s (see Algorithms 2 and 3). Hence, departure of nodes satisfies the invariant as well.

Algorithm 1 Join Algorithm at Node n

1: $d_n := succ_{\mathcal{P}}(n) \ominus n$
2: **if** $succ_{\mathcal{P}}(n) \neq n$ **then**
3: send <**join**, d_n > to $succ_{\mathcal{P}}(n)$
4: **end if**
5: BecomeNormal()

Algorithm 2 Leave Algorithm at Node n

1: **if** $succ_{\mathcal{P}}(n) \neq n$ **then**
2: send <**leave**, d_n > to $succ_{\mathcal{P}}(n)$
3: **end if**
4: $d_n := 0$

Algorithm 3 BecomeNormal algorithm at node n

```
1: while true do
2:     receive m from q
3:     if m=<join, v> then
4:         d_n := d_n − v
5:     else if m=<leave, v> then
6:         d_n := d_n + v
7:     end if
8: end while
```

Thereafter, we use an aggregation protocol to let each node in the system calculate the average inter-node distance. We use a slightly modified version of the averaging algorithm (AVG) by [**?**,**?**] to calculate the average inter-node distance.

The modified AVG algorithm is shown by Algorithm 4. It works by letting every node in the system periodically pick a random node which it performs an atomic exchange with. If a node i randomly picks node j, the exchange algorithm works as follows. Node i sends d_i to node j, which sets $d_j = \frac{d_j+d_i}{2}$. Node j then sends its updated d_j to node i which sets $d_i = d_j$. This algorithm converges in logarithmic number of steps as shown in [**?**], and every node n will eventually have $d_n = \sum_{i \in \mathcal{P}} \frac{d_i}{|\mathcal{P}|}$.

Algorithm 4 AVG algorithm at node n

```
1: Active Node
2: r := random({0, .., N − 1})
3: m := lookup(r)
4: send <exch, d_n> to m
5: receive <reply, i> from m
6: d_n := i

7: Passive Node
8: while true do
9:     receive <exch, i> from m
10:    d_n := (d_n+i)/2
11:    send <reply, d_n> to m
12: end while
```

So far, we have assumed that successor and predecessor pointers are correct, and that there are no failures in the midst of the asynchronous exchange taking place. In the subsequent sections, we will incrementally make the algorithm handle such situations.

3.3 Implementation and Failure Handling

In this section, we show how the algorithm can efficiently be deployed on-top of most structured peer-to-peer systems which provide a *Distributed Hash Table* (DHT) abstraction to handle crash failures. Hence, our scheme is general enough to be applied to any structured P2P system that provides a DHT abstraction.

A DHT abstraction is simply a hash table that is distributed and transparently shared on all the nodes in the system. Hence, any node can supply a key and find the value associated with that key in the globally distributed hash table.

The problem with the scheme that we outlined in the previous section is that the failure of a node will break the invariant that $N = \sum_{i \in \mathcal{P}} d_i$. The failure of node f implies that $N - d_f = \sum_{i \in \mathcal{P}} d_i$. Hence, we need to recover the value of d_f to restore the invariant.

In all structured peer-to-peer systems, according to our knowledge, a node with identifier i is responsible for identifier i. In other words:

$$(\exists i \in \mathcal{P}) \rightarrow (lookup(i) = i)$$

For our purposes, we let each node i store its estimate d_i in the distributed hash table under the key i with d_i as the value. Since node i is responsible for the key i, this operation is just a local operation which is fast and efficient.

The advantage of storing the estimate in the DHT is that it will be replicated automatically by the system as DHTs commonly have built-in replication[**?**,**?**,**?**]. Furthermore, DHTs automatically detect crash failures to fetch and restore key/value pairs that were stored on a crashed node. Whenever a node replicates a key/value pair representing an estimate as the result of a failure, it should add that value to its own estimate, to ensure that the invariant is restored. This is straight-forward in most DHTs and we have implemented it in our system[**?**].

The above scheme will work as long as the replicas are consistent, and there are no more than f failures in a system with f-replication degree.

4 Making the Algorithm Self-Stabilizing

In this section, we augment the network size estimation algorithm with self-stabilizing waves[**?**] to ensure that it can recover from arbitrary estimate values of d_i, hence situations when $N \neq \sum_{i \in \mathcal{P}} d_i$. We believe that the algorithm given in the previous sections is fast and efficient, but would diverge

indefinitely as soon as the invariant was broken a single time. Hence the need for self-stabilization.

The idea is to periodically send a token around the ring, which calculates the sum of d_i and adjusts the the sum to restore the invariant. As in other self-stabilizing algorithms, the system is asymmetric as one designated node behaves differently from the other nodes. In peer-to-peer systems, however, we have the additional burden of handling the permanent crash failure of the designated node. Our solution is to let the successor of identifier 0 be the designated node. I.e, node i is the designated node iff $succ_{\mathcal{P}}(0) = i$. This works as every node knows its successor and predecessor and can locally tell if it is the designated node. Should the designated node fail, the structured peer-to-peer system automatically detects this and reconnects the ring by using standard successor-lists[**?**].

To restore the invariant, at periodic time intervals, self-stabilizing waves are sent out with the purpose of finding out the sum of all current estimates. This is done by the designated node by sending a special message around the network containing the sum of the estimates of visited nodes. When the token, together with its sum σ, arrives back at the designated node, the designated node s updates its estimate to $d_s = d_s + N - \sigma$.

Algorithm 5 Self-stabilizing AVG algorithm at node n

1: *Active Node*
2: $r := random(\{0, .., N-1\})$
3: $m := lookup(r)$
4: send <**exch**, d_n, w > to m
5: receive <**reply**, i, x> from m
6: **if** $x > w$ and $save_x$ is undefined **then**
7: $\quad save_x := d_n$
8: **end if**
9: $d_n := i$

10: *Passive Node*
11: **while true do**
12: $\quad$ receive <**exch**, i, x> from m
13: $\quad$ **if** $x > w$ and $save_x$ is undefined **then**
14: $\quad\quad save_x := d_n$
15: $\quad$ **end if**
16: $\quad d_n := \frac{d_n+i}{2}$
17: $\quad$ send <**reply**, d_n, w> to m
18: **end while**

The above algorithm will, however, not work if there are exchanges happening during the token circulation. For example, assume a ring of size $N = 100$ with nodes $\mathcal{P} = \{15, 55, 85\}$ with estimates $20, 30, 50$, respectively. The designated node is node 15, which sends the token with $\sigma = 20$ to node 55, which adds its estimate 30 to σ. The token is then passed to node 85, but before node 85 receives the token, an averaging exchange takes place between nodes 85 and 55, which results in the estimates being $20, 40, 40$, respectively. Thereafter, node 85 receives the token and adds its estimate 40. When the token comes back to the designated node, it contains the sum 90, which is incorrect as the sum always has been 100.

Algorithm 6 Self-stabilizing Wave Running at Node n

```
1: Designated Node
2: if n = succ_P(0) then
3:     w := w + 1
4:     z := 0
5:     send <wave, w, d_n > to succ_P(n)
6:     receive <wave, w, σ> from pred_P(n)
7:     d_n := d_n + z + N − σ
8: else
9:     receive <wave, x, σ> from pred_P(n)
10:    if save_x is undefined then
11:        save_x := d_n
12:    end if
13:    send <wave, x, σ + save_x > to succ_P(n)
14:    w := x
15: end if
```

We remedy this by associating a number with each wave, which is increased for every new wave. To see how this works, assume that the system contains the nodes $A = \{n_1, n_2, \cdots, n_L\}$, and wave i sent by designated node n_1 has been circulated to all the nodes $\{n_1, \cdots, n_x\}$ where $x \leq L$. If two nodes i and j are to engage in a random exchange, where i has not been visited by the latest wave while j has been visited by the latest wave, the collected sum will be affected erroneously. Hence, our algorithm detects this situation by comparing the latest wave number the two interacting nodes have seen. Thus node j informs i such that i can save its estimate d_i in a temporary variable associated with wave i, making i report the saved value once the wave reaches it. Algorithm 5 shows the extended AVG algorithm running on node n with the variable w containing the count of the last seen wave. When wave with count i reaches a node n, the node checks to see if

it has a saved value $save_i$, in which case that value is added to the sum, otherwise d_n is added to the sum. Algorithm 6 shows the ring algorithm.

We will now show that the algorithm is self-stabilizing with respect to arbitrary estimates.

Lemma 1. *Given that no join, leave, failures, or corrupt estimates occur, a wave ensures that the actual sum of all estimates, d, at the moment the wave was initiated by the designated node, is reported to it at the end of the wave.*

Proof. Let $t \in \{0, \cdots, N\}$ denote the step in the wave algorithm, where $t = 0$ is when only the designated node knows about the wave, and $t = N$ when all nodes have been visited and the wave have reached the designated node again.

The nodes in the ring, A, have either been visited by the latest wave or not. Hence, at any step, t, the ring is partitioned into two sets V_t and $W_t = (A - V_t)$, representing those visited by the wave and those not visited by the wave. Let σ_t denote the sum collected by the wave at step t.

It can be shown by induction on the steps that the algorithm satisfies the invariant that $\sigma_\tau = \sum_{n \in W_\tau} d_n$ for all τ.

Two nodes $i, j \in V_t$ making an exchange will not affect the value in σ_t as those nodes values is already reflected in σ_t. Two nodes $i, j \in W_t$ making an exchange will not affect the value in σ_t either, as their sum before the exchange is equal to their sum after the exchange. I.e. $d_i + d_j = 2\frac{d_i + d_j}{2}$.

The remaining case, $i \in V_t$ and $j \in W_t$, preserves the invariant as well as j will save its estimate the first time it detects i is part of a more recent wave. The saved estimate is what will be reported when the wave reaches j. □

Similarly to exchanges happening during the wave, joins and leaves affect the sum collected by the wave. We now show how joins and leaves can be handled to ensure that the collected sum is correct.

Joins (see Algorithms 7, 9) are handled simply by letting the joining node j set its local estimate to $d_j = 0$, hence a join simply does not affect the collected sum. Furthermore, a joining node inherits the same wave number as its successor.

Leaves (see Algorithms 8, 9) are handled by letting their successor, s, save its old value d_s similarly as previously. Furthermore, if the successor is the designated node, it will add the leaving node's estimate in a special variable z which will adjust the collected sum.

Lemma 1 can now be easily be extended to joins and leaves. Hence, the algorithm is only prone to failures which would stop the wave from arriving at the designated node. We therefore let the designated node time out after sufficiently large time so that this problem can be solved. This would, however, mean that there might be more than one token in the ring at the same time. This could be solved by composing our algorithm with a self-stabilizing token passing algorithm for rings such as [**?**].

Algorithm 7 Self-stabilizing Join Algorithm at Node n

```
1: if succ_P(n) = n then
2:     d_n := N
3: else
4:     d_n := 0
5: end if
6: BecomeNormal()
```

Algorithm 8 Leave Algorithm at Node n

```
1: if succ_P(n) ≠ n then
2:     send <leave, d_n, w > to succ_P(n)
3: end if
4: d_n := 0
```

Algorithm 9 BecomeNormal algorithm at node n

```
1: while true do
2:     receive m from q
3:     if m=<leave, v, x> then
4:         if x > w and save_x is undefined then
5:             save_x := d_n
6:         end if
7:         if succ_P(0) = n then
8:             z := z + v
9:         end if
10:        d_n := d_n + v
11:    end if
12: end while
```

Theorem 1. *Convergence: The algorithm eventually makes the invariant* $N = \sum_{i \in \mathcal{P}} d_i$ *true.*

Proof. By the above lemma, the sum, s, reported at the end of the wave is the actual sum of all estimates at the moment the wave was initiated, given that the perturbations have ceased. As atomic exchanges do not change the sum, neither do joins and leaves, the sum of all estimates at the time the wave ends is equal to sum of the estimates when the wave started.

Since the designated node adjusts its local estimate by adding $N - s$ to it, the invariant becomes true. □

Theorem 2. *Closure: The invariant says true once it is realized and no failures and estimate corruptions occur.*

Proof. Similar to the convergence proof. Here, however, $s = 0$ always and the adjusting of the local estimate of the designated node does not affect the total sum, and hence the invariant is preserved. □

5 Related Work

Network size estimation algorithms have been extensively studied in different contexts in distributed systems. However, peer-to-peer systems add an extra level of complication as nodes may fail, and the network size varies dynamically over time, and the estimation algorithm needs to continuously update its estimation to reflect the number of nodes. Unfortunately, most of the algorithms for network size estimation in peer-to-peer networks are not self-stabilizing and give no guarantee that a forever-running system does not diverge indefinitely because of some corrupt estimates. The authors of [**?**,**?**] propose an algorithm where one node sets its estimate to 1 and the rest set their estimates to 0. Thereafter, the system computes the average of all the estimates, which yields $d_n = 1/p$ where p is the number of nodes in the system. The algorithm is however not robust to failures and leaves, as the node with value 1 might want to leave the system shortly after starting the algorithm, in which case it will have serious impact on the outcome of the estimate. The authors of [**?**] and [**?**] mention that a nodes distance to its successor can be used to calculate the number of nodes in the system, but provide no reasoning that the value always converges exactly to the correct value.

6 Conclusion

We have presented a self-stabilizing network size estimation gossip algorithm which determines the number of nodes in a structured peer-to-peer system. The algorithm handles joins, leaves, and failures and is applicable to most structured peer-to-peer systems providing a distributed hash table abstraction. Furthermore, the estimates of the nodes can be arbitrary wrong at any given time, the algorithm will eventually correct all estimates to the correct value, even while join and leaves and gossiping is taking place. The algorithm only assumes that the system is weakly fair, and does, hence, not require the nodes to make the same number of exchanges, to be correct. Furthermore, we proved its convergence and closure.